Dynamic Synthetic Weight Matrices for Neural Networks

Problem Statement:

When we look at **weight matrices** in current neural networks — especially dense (fully connected) layers — they come with a set of **computational and memory-related drawbacks** compared to something like a **Hadamard–Schur product** approach (simple element wise multiplication.)

Breaking it down into the main issues:

---

1. O(n²)Compute Cost

- **Dense layers** with an n×n weight matrix require n² multiplications per forward pass (plus n² additions).
- This quadratic scaling becomes expensive when n is large, especially in large language models or vision transformers.
- Even with GPUs, which can parallelize, the **energy cost** grows rapidly.

Contrast with Hadamard–Schur:

- A Hadamard (elementwise) product has **O(n)** cost — only one multiplication per element.
- No need for full matrix-matrix multiply kernels.

---

2. Multiply–Accumulate (MAC) Bottleneck

- Dense layers depend heavily on MAC operations, which are:
    - **Expensive in energy** compared to additions alone.

    - Limited by hardware's **MAC throughput** (often the main FLOP bottleneck in training).

- **Matrix multiply** requires reading the same weights many times, amplifying data movement cost.

Hadamard–Schur advantage:

- Only a **multiply**, no accumulation over rows/columns.
- No large dot products — lower precision formats can be used without stability issues.

---

3. Memory Access Patterns

- Dense matrix multiply requires **non-contiguous memory reads** for one of the operands (unless transposed), leading to:
    - Cache misses.
    - Strided memory access inefficiency.
- GPUs need large block-tiling schemes to get coalesced memory access.

Hadamard–Schur advantage:

- **Perfectly contiguous** memory access — read vector A, read vector B, multiply elementwise.
- Can be fused with activation functions in the same memory pass.

---

4. Weight Storage Cost

- An n×n dense weight matrix stores n^2 parameters.

- This can dominate model size — especially when compared to options involving structured transforms.

Hadamard–Schur advantage:

- Only O(n) parameters if the “weight” is a vector (diagonal matrix equivalent).
- Can reuse the same vector across multiple channels with negligible storage.

---

5. Poor Data Locality

- In GEMM (general matrix multiply), each input vector is multiplied by many weight elements spread across memory.
- For very large matrices, **weights can’t fit in cache**, leading to repeated DRAM fetches.
- DRAM access dominates energy cost in modern AI accelerators.

Hadamard–Schur advantage:

- Simple linear memory scan — high data reuse from cache, almost no indexing overhead.

---

6. Gradient Backpropagation Cost

- Backprop through dense matrices is **also $O(n^2)$** and requires two additional GEMMs (for input and weight gradients).
- This means training cost is **~3×** the forward cost.

Hadamard–Schur advantage:

- Backprop is also elementwise — just multiply by the same mask.

- Extremely cheap gradient computation.

---

7. Hardware Mismatch

- Many accelerators (e.g., TPUs, GPUs) are optimized for large block matrix multiplies, but these:
    - Consume a lot of power.
    - Require specialized scheduling and large SRAM buffers.
- On CPUs, large dense matrices often underperform due to **SIMD underutilization**.

Hadamard–Schur advantage:

- Matches SIMD and vectorized instructions perfectly.
- Fewer registers and smaller SRAM footprint.

---

8. Overparameterization Risk

- N^2 free parameters can easily lead to overfitting without strong regularization.
- Dense layers may learn redundant patterns.

Hadamard–Schur advantage:

- Lower parameter count forces the model to rely on **structural transformations** (e.g., Hadamard transforms before the elementwise product) for expressiveness.

Summary Table

Property	Dense Weight Matrix	Hadamard–Schur Product
Compute cost	$O(n^2)$	$O(n)$
MAC usage	Heavy	Minimal
Memory access	Strided / tiled	Sequential
Storage	$O(n^2)$	$O(n)$
Backprop cost	High	Low
Hardware friendliness	Needs large GEMM units	Perfect for SIMD/streaming
Overfitting risk	Higher	Lower

If you combine a **fast transform** (Hadamard, FFT, DCT) before and after a **Hadamard–Schur product**, you can get an **expressive transform with O(nlogn)** compute instead of $O(n^2)$ — this is essentially the idea behind *structured efficient layers*.

Structured Dense Matrices:

1. Introduction to Structured Weight Matrices

In many machine learning systems, weight matrices are **dense** — meaning that every input can directly influence every output. Dense matrices are powerful because they allow *full connectivity*, but they have some downsides:

- **High computation cost**: multiplying an n×n dense matrix by a vector costs $O(n^2)$ operations.
- **Large memory footprint**: $O(n^2)$ storage.
- **Poor cache locality**: irregular memory access patterns.

Structured weight matrices aim to keep much of the expressiveness of dense matrices while reducing compute and memory cost. They impose a mathematical structure (often derived from a known fast transform) so multiplication can be done in O(nlogn) instead of $O(n^2)$.

---

2. One-to-All Connectivity in Dense and Fast Transforms

If you multiply a vector x by a **dense** matrix M:

$$y=Mx$$

a change in *any single* element of x can potentially change *all* elements of y.
This is **one-to-all connectivity** — every input affects every output.

Fast transforms like the **Walsh–Hadamard transform (WHT)** or **Fourier transform (FFT)** also have this property.
For example, if H is the n×n Hadamard matrix:

$$y=Hx$$

then any change in one element of x changes **all elements** of y, even though H can be applied in O(nlogn) operations.

This means:

> A fast transform is **functionally dense** (one-to-all mapping) but computationally structured.

---

3. Fast Transform as a "Density Amplifier"

Consider a sparse matrix S (most entries zero). If we surround it with fast transforms:

$$M=HSH$$

then M becomes **dense** even though S is sparse.
Why?

- The first H spreads local input changes across all intermediate variables.
- The sparse S makes changes in multiple intermediate places.
- The second H mixes them again into all outputs.

This trick is used in **Fastfood transforms** and **butterfly matrices** in machine learning.

---

4. Matrix Equivalents of Fast Transforms

Even though we apply the WHT or FFT algorithmically, each has a **matrix equivalent**:

- **Walsh–Hadamard**: a real ±1 matrix H of size n×n.
- **Discrete Fourier Transform (DFT)**:
    - Can be represented as a **cosine matrix** and **sine matrix** (real-valued),
    - Or combined into a single **complex-valued matrix** F.

These matrices are *mathematically dense* and *invertible*, even though their fast algorithms avoid explicitly storing them.

---

5. Adjustable Dense Mapping Using H and a Scaling Vector

Instead of S being a full matrix, let it be a **single column vector** s of length n.
Given an input vector x:

1. Apply the fast transform:
   $u = Hx$
2. Apply **elementwise scaling** (vector form of Schur–Hadamard product):
   $v = s \circ u$
   where $\circ$ means elementwise multiplication.
3. Optionally apply another fast transform:
   $y = Hv$

The full operation can be written as:

$y = H \, \mathrm{diag}(s) \, Hx$

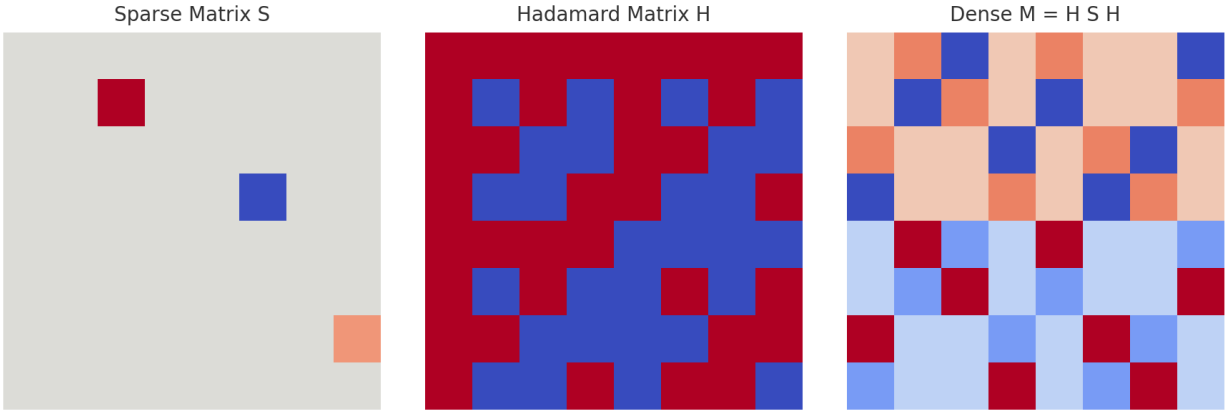
Here, diag(s) is a diagonal matrix with s on its diagonal — but we never need to form it explicitly.

Advantages:

- Computation is O(nlogn) from the two Hadamard transforms.
- The scaling step is O(n), very SIMD-friendly (ideal for GPUs, vectorized CPU code).
- Retains **dense mapping behavior** because H provides one-to-all mixing before and after scaling.

---

7. Approximation of Arbitrary Dense Mappings



While a single H diag(s) H mapping can't represent every dense matrix exactly, stacking multiple such layers with different s vectors can approximate arbitrary dense matrices.
This is similar to how multiple convolutional or recurrent layers build complex representations.

---

8. Role of Redundant Information in the Input

Since H spreads information globally but scaling s operates elementwise, performance improves if there is **redundant information** in x. Redundancy makes it easier for the scaling step to select and modulate the right components before the second mixing.

---

9. Increasing Redundancy via Vector Copying

A simple redundancy boost method:

1. **Copy** x multiple times into a larger vector x′.
   Example: x′=[x,x,x,x] (length m=k⋅n).

2. Apply H diag(s) H to x′.

3. Reduce back to size n by **subsampling**, **averaging**, or **summing** groups of outputs.

This keeps the benefits of the H diag(s) H structure while increasing the effective degrees of freedom in the transformation.

Summary Flow

1. **Dense matrices** → one-to-all connectivity, expensive $O(n^2)$.

2. **Fast transforms** → also one-to-all connectivity, but cheap O(nlogn).

3. **Fast transform sandwich** → sparse becomes dense.

4. **Walsh / Fourier equivalents** → explicit matrix forms exist.

5. **Schur–Hadamard product** → adjustable dense-like matrix with fast computation.

6. **Approximation power** → multi-layered can match arbitrary dense matrices.

7. **Redundancy** → improves mapping performance.

8. **Vector copying trick** → simple redundancy injection + subsampling.

Dynamic Structured Weight Matrices:

1. Starting Point — Random Projections with Fast Transforms

One efficient way to transform data before it enters a neural network is via a **random projection**:

- Multiply input x by a **random diagonal sign-flip matrix** D (entries are ±1, fixed after initialization).

- Apply a **fast transform** like the Walsh–Hadamard transform H:
  $z=H(Dx)$
- Optionally, subsample or scale z to match the desired feature dimension.

Why this is useful:

- The sign flips scramble the data in a way that makes the Hadamard transform act more like a random projection.

- D and H together create a **dense one-to-all mixing** while keeping cost at $O(n\log n)$.

- This method is used in **Fastfood transforms** and some random feature maps for kernels.

The random projection above is **static** — it's the same transformation for every input.
We can make it **data-dependent** by:

- Using **features of the input** to choose which weights, neurons, or sub-blocks of a structured weight matrix are active.

- This turns the weight matrix into something **dynamically constructed per input**.

---

2. Introduction to Weight Switching from Data Features

Weight switching is the idea of enabling or disabling certain weights in a neural network depending on the input.
The control signal for switching can come from:

- **Direct data features** (values or patterns from the raw input).

- **Derived features** computed from the data, including random features such as those obtained from H(Dx) or other projections.

By using these features to gate weights or blocks of weights, the network's effective connectivity changes per input, allowing more adaptive and context-sensitive computation.

3. Neural Networks Already Have Implicit Dynamic Weights

Even a conventional dense ReLU neural network already exhibits **dynamic weight selection**:

- In a layer:
  $y=\text{ReLU}(Wx+b)$
- If a ReLU output is **zero**, the forward-connected weights from that neuron to the next layer contribute nothing to the final output.

- In effect, the **columns in the next layer's weight matrix** connected to inactive neurons are **dropped** for that input.

- The set of "active" weights changes for each input, so the **effective weight matrix is different for each example**.

This is a form of **input-dependent sparsity**.

---

4. Explicit Feature-Based Weight Switching

Instead of relying solely on ReLU's implicit pruning, we can design systems where:

- Certain **input features** control which weights are **enabled**.

- These "control features" could be:

    - Original data features (e.g., raw sensor values, pixel intensities).

    - Random projections of the data (e.g., $H(Dx)$).

    - Features from earlier layers in the network.

    - Binary thresholded versions of features (gating signals).

---

5. What to Switch: Possible Granularities

Feature-driven selection can control weights at different levels of granularity:

1. **Individual weights**
    - Each connection weight has an enable/disable flag determined by the features.
    - Fine-grained, but potentially high control complexity.
2. **Rows of a weight matrix** (input-to-neuron weights)
    - Equivalent to enabling/disabling entire neurons in a layer.
3. **Columns of a weight matrix** (forward connections from a neuron)
    - Controls how much influence a neuron has on the next layer.
4. **Blocks in structured weight matrices**
    - For fast transforms, you might enable/disable certain butterfly stages or submatrices.
5. **Whole weight matrices**
    - Switching between different pre-trained matrices depending on input type.

---

6. Structured Matrices with Dynamic Selection

Structured weight matrices like:

- H diag(s) H
- FFT-based blocks
- Circulant or Toeplitz matrices

can also be dynamically modified by:

- Making s depend on input features (e.g., computed by a small auxiliary network).
- Selecting **subsets of elements in s** based on features (binary gating).
- Switching between different structured blocks based on features.

Because the structure is known (like H), we can:

- Change only a small vector s instead of a full n×n matrix.
- Retain O(nlogn) efficiency.

---

7. Example — Feature-Driven Gating in a Structured Matrix

A practical example:

1. Compute a feature vector f from the input (e.g., f=H(Dx) or early-layer activations).
2. Binarize f to produce a binary mask m.
3. Apply m to the scaling vector s in H diag(s) H, enabling only selected components.
4. Use the resulting dynamically modified matrix to transform the main input or an intermediate representation.

Extension — Selecting from a Large Pool of Weights
Instead of having a single s vector, maintain a **large pool of candidate scaling vectors** $\{s_1, s_2, \ldots, s_{\square}\}$ or even more complex parameter sets.

- Binary features derived from f act as **indices or selectors** to choose which subset of these parameters to use for the current input.
- This lets the network maintain **multiple specialized structured weight matrices** and select among them on-the-fly.
- In effect, the network's **capacity** grows without requiring all weights to be active at once — only the relevant subset is loaded and applied for each input.

This idea is similar to a **mixture-of-experts** model but applied at the level of **dynamically structured weight matrices** rather than entire subnetworks.

---

8. Why This Matters

- Can lead to **conditional computation** — only parts of the network run for a given input.
- Improves efficiency and potentially accuracy by specializing parts of the network to certain input types.
- Extends naturally from standard dense networks to structured, fast-transform-based networks.

Dynamic Structured Weight Matrices Without Explicit Non-Linearity

1. Usual reason for non-linear activations

In standard neural networks, non-linear activation functions (like ReLU, sigmoid, tanh) are essential because:

- A stack of purely linear layers $y=Wx$ collapses into a single equivalent linear transformation.
- Without non-linearity, the network cannot approximate arbitrary non-linear functions.

---

2. How dynamic structured weight matrices change the game

A **dynamic structured weight matrix** is **not fixed** — it changes depending on the input.
If we write it as:

$$W(x)=H\,\mathrm{diag}(s(x))\,H$$

then $s(x)$ is a scaling vector computed from the **same input** x or from derived features $f(x)$.

This means:

$y = W(x) \cdot x$

is **not** a purely linear map anymore, because W depends on x.
The multiplication is still element-wise linear in x **for a fixed W**, but since W changes with x, the **overall mapping is non-linear**.

---

3. Why you can skip explicit non-linear activations

- If s(x) is computed from x in a non-trivial way (e.g., thresholding, random feature maps, or other feature extractors), the element-wise multiplication W(x)x already introduces **input-dependent multiplicative interactions**.

- These multiplicative interactions give the same functional effect as non-linear activations: they allow the network to combine and transform inputs in ways a fixed linear system cannot.

Example:
If s(x) contains binary gates from features of x, the output neurons are switched on or off in a data-dependent way — similar to how ReLU zeroes certain neurons.

---

4. Including non-linear activations is optional

- You can still place a ReLU, GELU, or other non-linear function after W(x)x if you want **extra flexibility**.

- But even without it, the dynamic nature of W(x) ensures the mapping is non-linear in x.

---

5. Summary

- In static weight networks: non-linear activations are mandatory for non-linearity.

- In **dynamic structured weight networks**: the input-dependent nature of the weights already **builds in non-linearity**.

- Explicit activations are optional, not essential, though they can still help with training dynamics or representation shaping.

A Specific Dynamic Structured Weight Matrix Neural Network

Input Processing

The input vector x (which may be formed by replicating a smaller base input x′) is first passed through a **fixed random pattern of sign flips**. This ensures variation across dimensions and helps remove bias.
Next, the vector is transformed using the **fast Walsh–Hadamard transform (FWHT)**. This serves two purposes:

1. It spreads information evenly across dimensions (a form of random projection).

2. It avoids **spectral bias**, which would otherwise cause the FWHT alone to concentrate energy in only a few coordinates.

Layer Structure

At the start of each layer:

1. A fixed number of elements from the transformed input are **sampled** and **binarized**.

2. Each resulting binary feature bit selects between two candidate weight vectors.

3. All the selected weight vectors are **summed element-wise**, producing a single **dynamic scaling vector** S.

4. The layer then applies the structured transform:
   $y = H\,\mathrm{diag}(S)\,Hx$
   where H is the Walsh–Hadamard transform matrix.

This “sandwich” form — $H\,\mathrm{diag}(S)\,H$ — ensures that each dynamic scaling vector effectively creates a dense, input-dependent weight matrix.

Network Depth

- Layers can be stacked (“telescoped”) using the same pattern, providing deeper representations.

- At the final stage, a second fixed random sign-flip pattern is applied to the network output, ensuring symmetry with the randomized preprocessing.

Backward Pass

During backpropagation, the error signal encounters the same structural elements in reverse:

- A sign flip, followed by an FWHT, produces a **backward random projection** of the error.

- This allows gradients to flow through the network consistently with the structured transforms.

Properties and Advantages

- **Similarity to Dense ReLU Networks:**
  Like conventional dense ReLU networks, this architecture has relatively few switching choices per parameter (fractions of a percent). This favors smooth interpolation over brittle switching.

- **Efficient Operations:**
  All computations reduce to simple element-wise additions and Schur–Hadamard products, which map well to **streaming computation** on GPUs and CPU SIMD, with efficient memory caching and prefetching.

- **Implementation Note:**
  While the earliest stages of the FWHT algorithm involve somewhat awkward indexing (a known issue for GPU/CPU implementations), the overall transform is extremely fast and well-suited to large-scale parallel hardware.

Java Code Example:

```
public final class SyntheticNet {
// MCG random number seed must be odd
        final static int mcgSeed=123455;
// MCG random number multiplier
```

```
        final static int mcgMul=0x93D765DD;
// MCG random number inverse multiplier
   final static int mcgMulInv=0x7AAE1A75;
// Length of the input and output vectors
// Must be a power of 2 like 8, 16, 32,....
    final int vecLen;
// Internal vector length after replication of the input vector
    final int n;
// The number of S vector pairs to sum together
    final int density;
// The depth of the neural network
    final int depth;
// Weight storage
    final float[] wts;
// Stores the input to each layer and the vector S (for training)
    final float[] sur;
// Working memory
    final float[] work;
// Working S vector
    final float[] synthetic;
// The current state of the MCG random number generator
    int mcgState;
// The training rate
    float rate;
// Constructor
// vecLen: Must be a power of 2 like 8,16,32... to fit in with the
// fast Walsh Hadamard transform.
// widthBy: Must be 1,2,4,... number of replications of input vector
// density: Number of binary selectors per layer to construct S vector
// depth: The depth of the neural network
// rate: Training rate
    public SyntheticNet(int vecLen, int widthBy,int density, int depth, float rate) {
        this.vecLen = vecLen;
        n=vecLen*widthBy;
        this.depth=depth;
        this.density=density;
        this.rate=rate/(float)Math.sqrt(density);
        wts = new float[2*n*density*depth];
        sur=new float[2*n*depth];
        work=new float[n];
        synthetic=new float[n];
        mcgState=-mcgSeed;
        float scale=8.0654e-10f/(float)Math.sqrt(density);
// Fill the weights array with suitable scaled random numbers
```

```
// from the Gaussian distribution
      for (int i = 0; i < wts.length; i+=n) {
         for(int j=0; j<n; j++) work[j]=(mcgState*=mcgMul)*scale;
// Central limit theorem the uniform distribution (+-) combinations
         WHT.whtN(work); // patterns of (+-) combinations
         System.arraycopy(work,0,wts,i,n);
      }
   }

   public void recall(float[] result, float[] input) {
// Set random number seed
                  mcgState=mcgSeed;
// Replicate up the input vector combined with sign flipping
      for (int i=0; i<n; i+=vecLen) {
                        for(int j=0;j<vecLen;j++){
                              work[i+j]=mcgState<0? -input[j]:input[j];
                              mcgState*=mcgMul;
                        }
      }
// Input random projection to mix input data while avoiding
// Spectral concentration
      WHT.whtN(work);
// For each layer
      for (int i=0; i < depth; i++) {
// Clear the S vector (synthetic)
         java.util.Arrays.fill(synthetic,0f);
// For the number of binary selection features
         for(int j=0;j<density;j++){
// Get the element
                              float x=work[j];
// Binarize x and get the weight index for 1 of 2 vectors
                              int idxW=(i*density+j)*2*n+(x<0f? 0:n);
// Sum the chosen weights into the S (synthetic) vector
                              for(int k=0;k<n;k++) synthetic[k]+=wts[idxW+k];
                     }
// If training keep the input to the layer and the constructed
// S vector
                     if(result==work){
                              System.arraycopy(work,0,sur,2*i*n,n);
                              System.arraycopy(synthetic,0,sur,2*i*n+n,n);
                        }
// Schur-Hadamard product the work vector with the S vector
                     for(int j=0;j<n;j++) work[j]*=synthetic[j];
                     WHT.whtN(work);
```

```
                }
// Do a final fixed pattern of sign flips
                for(int i=0;i<vecLen;i++){
                        result[i]=mcgState<0? -work[i]:work[i];
                        mcgState*=mcgMul;
                }
    }

// Training algorithm
    public void train(float[] target, float[] input) {
// Recall (forward pass)
        recall(work,input);
// Calculate the error and undo the final recall pattern of
// sign flips
        for (int i=vecLen-1; i>=0; i--) {
                        mcgState*=mcgMulInv;
                        float t=(target[i]-work[i])*rate;
            work[i]=mcgState<0? -t:t;
        }
// Because of replication you need to fill the upper part of
// the work array with zero
        java.util.Arrays.fill(work,vecLen,n,0f);
// Working backwards through each layer
        for (int i = depth-1; i >=0; i--) {
// The fast Walsh Hadamard transform is self-inverse
// (Self-undoing) So you just need to apply it again to
// backpropagate through it.
            WHT.whtN(work);
// Get the index for the stored input to this layer
// and also the stored S vector.
            int idxS=i*2*n;
// Update the selected weights using the layer input and
// the error vector
            for(int j=0;j<density;j++){
                                float x=sur[idxS+j];
                                int idxW=(i*density+j)*2*n+(x<0f? 0:n);
                                for(int k=0;k<n;k++) wts[idxW+k]+=work[k]*sur[idxS+k];
                        }
// Backpropagate the error vector which is as easy as forming the
// Schur-Hadamard product with the S vector.
                        for(int j=0;j<n;j++) work[j]*=sur[idxS+n+j];
        }
    }
```

```
    public static void main(String[] args) {
// Example usage
// input output vector length=256
// 4 by replication
// 15 binary features
// 3 layers
// 0.05 training rate
        SyntheticNet net=new SyntheticNet(256,4,15,3,0.05f);
// Sine Cosine training data
        float[][] cos=new float[256][256];
        float[][] sin=new float[256][256];
        float[] work=new float[256];
        for(int i=0; i<256; i++) {
            for(int j=0; j<256; j++) {
                double t=j*(0.03+0.0002*i);
                cos[i][j]=(float)Math.cos(t);
                sin[i][j]=(float)Math.sin(t);
            }
        }
// Train and display error
        for(int i=0; i<10000; i++) {
            float e=0f;
            for(int k=0; k<256; k++) {
                net.train(cos[k],sin[k]);
            }
            for(int k=0; k<256; k++) {
                net.recall(work,sin[k]);
                for(int j=0; j<256; j++) {
                    float diff=work[j]-cos[k][j];
                    e+=diff*diff;
                }
            }
            System.out.println("Error: "+e);
        }

    }
}


// Fast Walsh Hadamard transform algorithm optimized for Java.
// Much simpler code is possible.
// Input vector must be a power of 2 like 8,16,32...
// The transform is self-inverse
public class WHT {
```

```
public static void wht(float[] x) {
    final int n=x.length;
    if(n<=2048) {
        hstepTo8(x,0,n);
        if(n==8) return;
        hstep16(x,0,n);
        if(n==16) return;
        hstepN(x,0,n,16);
        if(n==32) return;
        hstepN(x,0,n,32);
        if(n==64) return;
        hstepN(x,0,n,64);
        if(n==128) return;
        hstepN(x,0,n,128);
        if(n==256) return;
        hstepN(x,0,n,256);
        if(n==512) return;
        hstepN(x,0,n,512);
        if(n==1024) return;
        hstepN(x,0,n,1024);
        return;
    } else {
        for(int i=0; i<n; i+=2048) {
            hstepTo8(x,i,i+2048);
            hstep16(x,i,i+2048);
            hstepN(x,i,i+2048,16);
            hstepN(x,i,i+2048,32);
            hstepN(x,i,i+2048,64);
            hstepN(x,i,i+2048,128);
            hstepN(x,i,i+2048,256);
            hstepN(x,i,i+2048,512);
            hstepN(x,i,i+2048,1024);
        }
        hstepN(x,0,n,2048);
        if(n==4096) return;
        hstepN(x,0,n,4096);
        if(n==8192) return;
        hstepN(x,0,n,8192);
        if(n==16384) return;
        hstepN(x,0,n,16384);
        if(n==32768) return;
        hstepN(x,0,n,32768);
        if(n==65536) return;
        hstepN(x,0,n,65536);
```

```
        if(n==131072) return;
        hstepN(x,0,n,131072);
        if(n==262144) return;
        hstepN(x,0,n,262144);
        if(n==524288) return;
        hstepN(x,0,n,524288);
        if(n==1048576) return;
        for(int hs=1048576; hs<n; hs+=hs) {
            hstepN(x,0,n,hs);
        }
    }
}


public static void whtN(float[] x) {
    final int n=x.length;
    final float scale=1f/(float)Math.sqrt(n);
    if(n<=2048) {
        hstepTo8(x,0,n,scale);
        if(n==8) return;
        hstep16(x,0,n);
        if(n==16) return;
        hstepN(x,0,n,16);
        if(n==32) return;
        hstepN(x,0,n,32);
        if(n==64) return;
        hstepN(x,0,n,64);
        if(n==128) return;
        hstepN(x,0,n,128);
        if(n==256) return;
        hstepN(x,0,n,256);
        if(n==512) return;
        hstepN(x,0,n,512);
        if(n==1024) return;
        hstepN(x,0,n,1024);
        return;
    } else {
        for(int i=0; i<n; i+=2048) {
            hstepTo8(x,i,i+2048,scale);
            hstep16(x,i,i+2048);
            hstepN(x,i,i+2048,16);
            hstepN(x,i,i+2048,32);
            hstepN(x,i,i+2048,64);
            hstepN(x,i,i+2048,128);
```

```
            hstepN(x,i,i+2048,256);
            hstepN(x,i,i+2048,512);
            hstepN(x,i,i+2048,1024);
        }
        hstepN(x,0,n,2048);
        if(n==4096) return;
        hstepN(x,0,n,4096);
        if(n==8192) return;
        hstepN(x,0,n,8192);
        if(n==16384) return;
        hstepN(x,0,n,16384);
        if(n==32768) return;
        hstepN(x,0,n,32768);
        if(n==65536) return;
        hstepN(x,0,n,65536);
        if(n==131072) return;
        hstepN(x,0,n,131072);
        if(n==262144) return;
        hstepN(x,0,n,262144);
        if(n==524288) return;
        hstepN(x,0,n,524288);
        if(n==1048576) return;
        for(int hs=1048576; hs<n; hs+=hs) {
            hstepN(x,0,n,hs);
        }
    }
}

static          void hstepTo8(final float[]x,int start,int stop) {
    for(; start<stop; start+=8) {
        float a=x[start];
        float b=x[start+1];
        float c=x[start+2];
        float d=x[start+3];
        float h0=a+b+c+d;
        float h1=a-b+c-d;
        float h2=a+b-c-d;
        float h3=a-b-c+d;
        a=x[start+4];
        b=x[start+5];
        c=x[start+6];
        d=x[start+7];
        float g0=a+b+c+d;
        float g1=a-b+c-d;
```

```
        float g2=a+b-c-d;
        float g3=a-b-c+d;
        x[start]=h0+g0;
        x[start+1]=h1+g1;
        x[start+2]=h2+g2;
        x[start+3]=h3+g3;
        x[start+4]=h0-g0;
        x[start+5]=h1-g1;
        x[start+6]=h2-g2;
        x[start+7]=h3-g3;
    }
}

static          void hstepTo8(final float[]x,int start,int stop,float scale) {
    for(; start<stop; start+=8) {
        float a=x[start];
        float b=x[start+1];
        float c=x[start+2];
        float d=x[start+3];
        float h0=a+b+c+d;
        float h1=a-b+c-d;
        float h2=a+b-c-d;
        float h3=a-b-c+d;
        a=x[start+4];
        b=x[start+5];
        c=x[start+6];
        d=x[start+7];
        float g0=a+b+c+d;
        float g1=a-b+c-d;
        float g2=a+b-c-d;
        float g3=a-b-c+d;
        x[start]=(h0+g0)*scale;
        x[start+1]=(h1+g1)*scale;
        x[start+2]=(h2+g2)*scale;
        x[start+3]=(h3+g3)*scale;
        x[start+4]=(h0-g0)*scale;
        x[start+5]=(h1-g1)*scale;
        x[start+6]=(h2-g2)*scale;
        x[start+7]=(h3-g3)*scale;
    }
}

static          void hstep16(final float[] x,int start,int stop) {
    for(; start<stop; start+=16) {
```

```
            float a=x[start];
            float b=x[start+1];
            float c=x[start+2];
            float d=x[start+3];
            float e=x[start+4];
            float f=x[start+5];
            float g=x[start+6];
            float h=x[start+7];
            x[start]+=x[start+8];
            x[start+1]+=x[start+9];
            x[start+2]+=x[start+10];
            x[start+3]+=x[start+11];
            x[start+4]+=x[start+12];
            x[start+5]+=x[start+13];
            x[start+6]+=x[start+14];
            x[start+7]+=x[start+15];
            x[start+8]=a-x[start+8];
            x[start+9]=b-x[start+9];
            x[start+10]=c-x[start+10];
            x[start+11]=d-x[start+11];
            x[start+12]=e-x[start+12];
            x[start+13]=f-x[start+13];
            x[start+14]=g-x[start+14];
            x[start+15]=h-x[start+15];
        }
    }

    static void hstepN(final float[] x,int start,int stop,int hs) {
        while (start < stop) {
            int j = start + hs;
            while (start < j) {
                float a = x[start];
                float b = x[start + hs];
                x[start] = a + b;
                x[start + hs] = a - b;
                start += 1;
            }
            start += hs;
        }
    }
}
```

The End.